

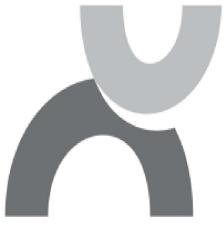


Programação Avançada

Hoje: Padrões de Projeto

Prof. Dr. Rafael P. Torchelsen
rafael.torchelsen@inf.ufpel.edu.br

O que é um padrão de projeto?



- Padrões de projeto são **soluções típicas para problemas comuns** no desenvolvimento de software.
- São como **modelos pré-fabricados** que você pode personalizar para resolver um **problema recorrente no seu código**.
- **Você não pode simplesmente encontrar um padrão e copiá-lo para o programa**, como faria com funções ou bibliotecas prontas.
- Padrões são frequentemente **confundidos com algoritmos**, porque alguns conceitos descrevem soluções típicas para problemas conhecidos.
- Enquanto um algoritmo sempre define um conjunto claro de ações que podem atingir um objetivo, um **padrão é uma descrição mais geral de uma solução**.

Do que consiste um padrão?



- A maioria dos padrões é descrita de forma muito formal para que as pessoas possam reproduzi-los em diversos contextos
- A **intenção** do padrão descreve brevemente o problema e solução.
- A **motivação** explica mais detalhadamente o problema e a solução que o padrão possibilita.
- A **estrutura** de classes mostra cada parte do padrão e como elas se relacionam.

História dos padrões

- Padrões de projeto não são conceitos obscuros e sofisticados.
- Padrões são soluções típicas para problemas comuns em design orientado a objetos.
- Quando uma solução se repete diversas vezes em vários projetos, alguém eventualmente a nomeia e a descreve em detalhes. É basicamente assim que um padrão é descoberto.
- O conceito de padrões foi descrito pela primeira vez por Christopher Alexander em **A Pattern Language: Towns, Buildings, Construction**. O livro descreve uma "linguagem" para projetar o ambiente urbano. As unidades dessa linguagem são os padrões. Eles podem descrever a altura ideal das janelas, o número de andares de um edifício, o tamanho ideal das áreas verdes em um bairro e assim por diante.
- A ideia foi retomada por quatro autores: Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm. Em **1994**, eles publicaram **Design Patterns: Elements of Reusable Object-Oriented Software**, no qual aplicaram o conceito de padrões de projeto à programação. O livro apresentava **23 padrões** que resolviam diversos problemas de design orientado a objetos e rapidamente se tornou um best-seller.
- Desde então, dezenas de outros padrões orientados a objetos foram descobertos. A "abordagem por padrões" tornou-se muito popular em outras áreas da programação, de modo que muitos outros padrões existem agora também fora do design orientado a objetos.

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M&T Books / Addison Wesley - Reading, Massachusetts. All rights reserved.

Foreword by Grady Booch



Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Holland. All rights reserved.

Foreword by Grady Booch



https://en.wikipedia.org/wiki/Design_Pattern

Material

DESIGN PATTERNS



Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

What's a design pattern?

Benefits of patterns

Patterns are a toolkit of solutions to common problems in software design. They define a common language that helps your team communicate more efficiently.

More about the benefits »

Classification

Design patterns differ by their complexity, level of detail and scale of applicability. In addition, they can be categorized by their intent and divided into three groups.

More about the categories »

Catalog of patterns

List of 22 classic design patterns, grouped by their intent.

Look inside the catalog »

Criticism of patterns

Are patterns as good as advertised?
Is it always possible to use them?
Can patterns sometimes be harmful?

More about the criticism »

Dive into DESIGN PATTERNS

Check out our ebook on design patterns and principles. It's available in PDF/ePub/Mobi formats and includes the archive with code examples in Java, C/C++, PHP, Python, Ruby, Go, Swift, & TypeScript.

Learn more about the book

History of patterns

Who invented patterns and when?
Can you use patterns outside software development? How do you do that?

More about the history »



REFACTORING . GURU.

Premium Content

Refactoring

Design Patterns

What is a Pattern

Catalog

Creational Patterns

Structural Patterns

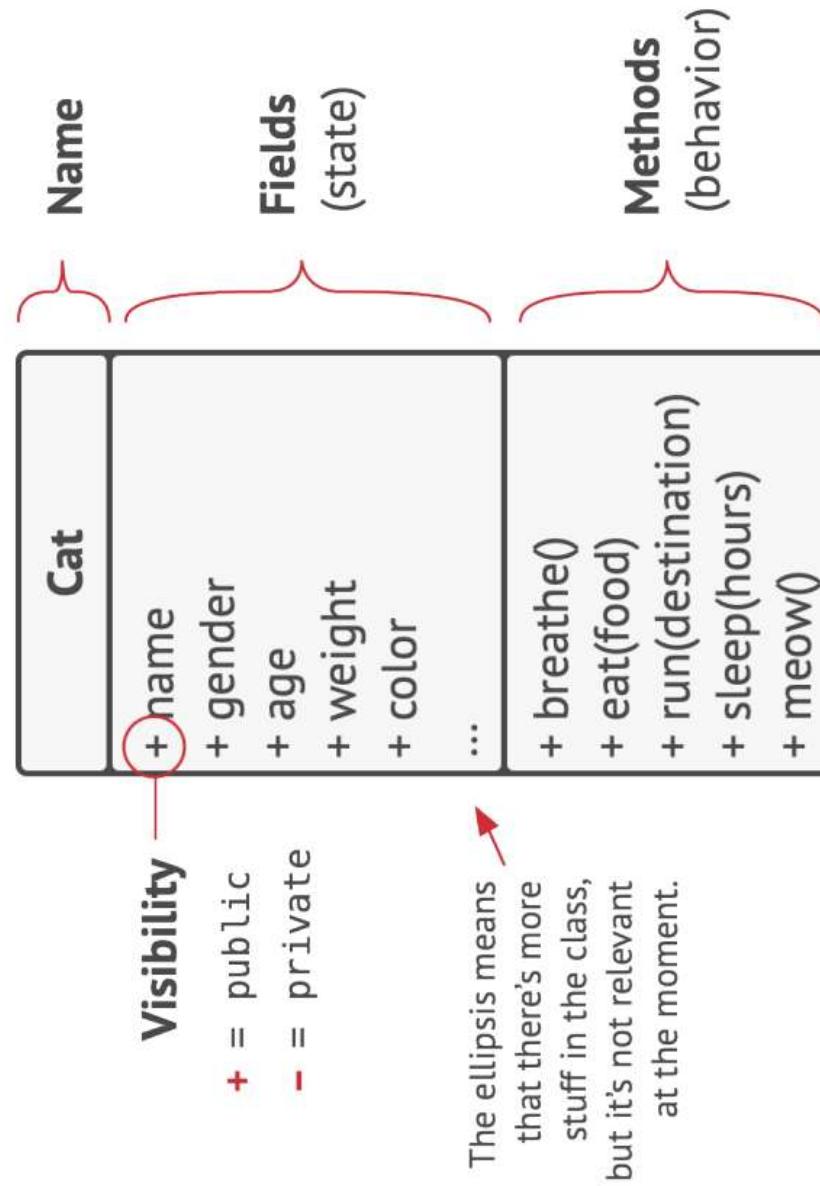
Behavioral Patterns

Code Examples

Sign in Contact us

<https://refactoring.guru>

Básico de Orientação a objetos



This is a UML class diagram. You'll see a lot of such diagrams in the book.

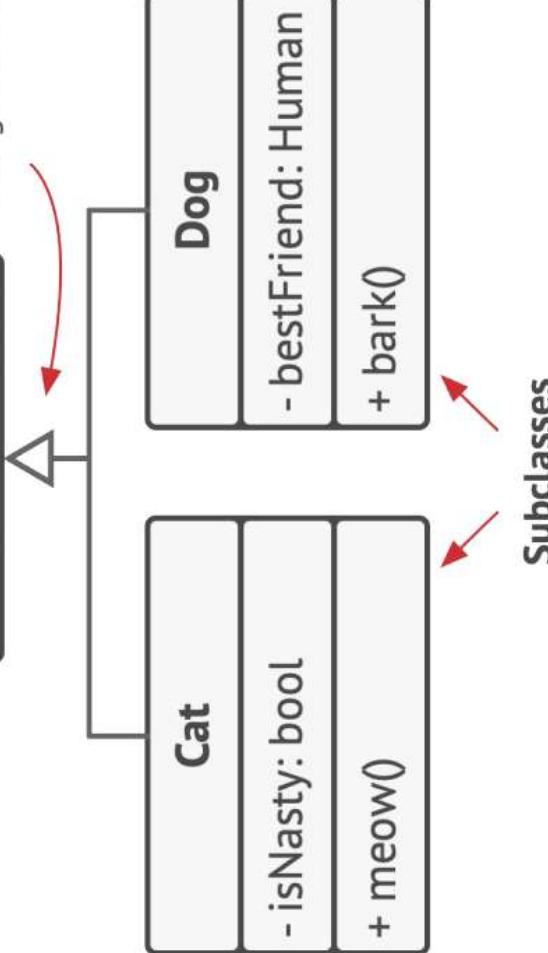
Hierarquía

Animal
Superclass

- + name
- + sex
- + age
- + weight
- + color

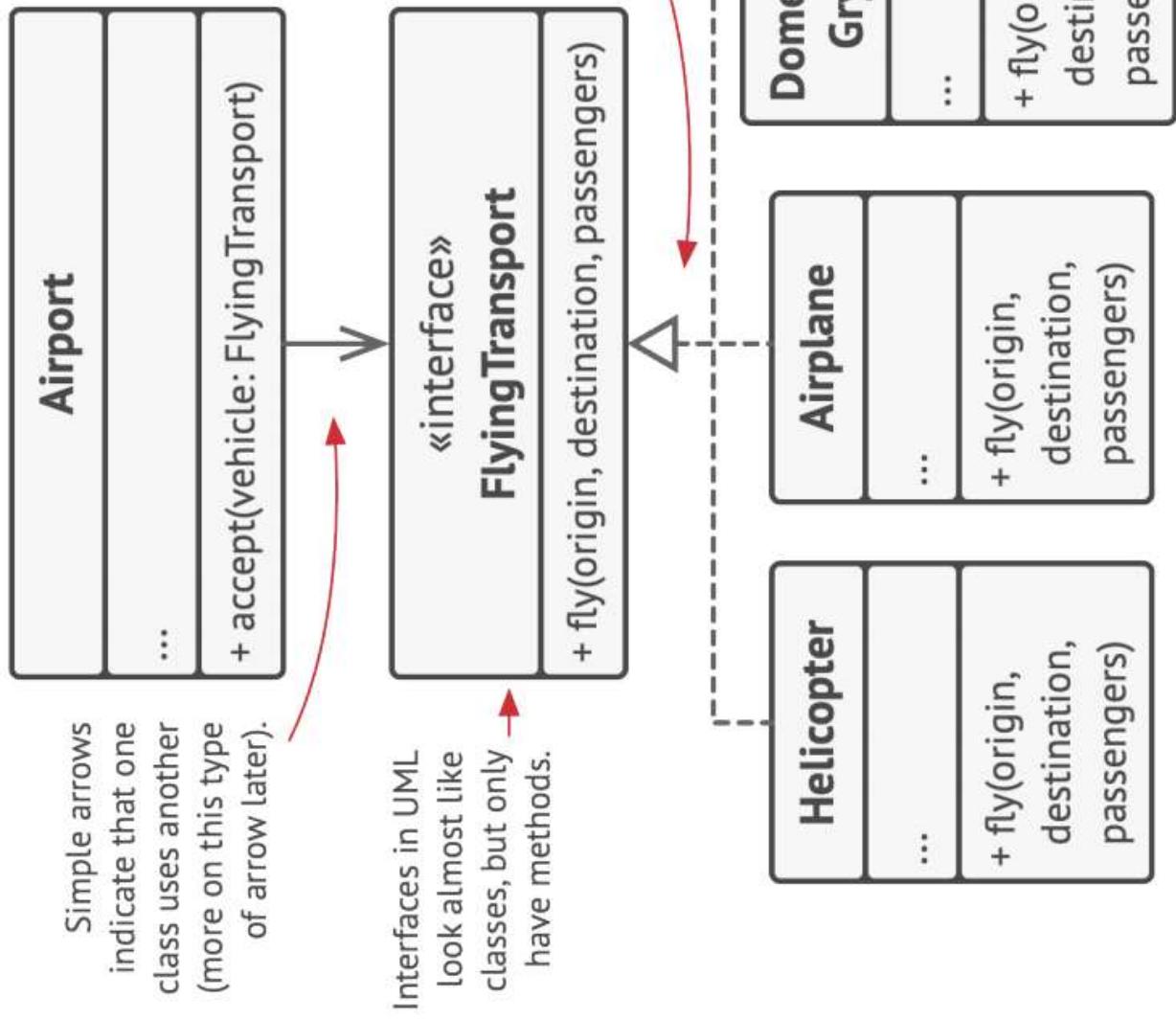
- + breathe()
- + eat(food)
- + run(destination)
- + sleep(hours)

Arrows with empty triangle heads indicate inheritance and always go from a subclass to a superclass.
Arrows from several subclasses can overlap (as in this diagram) or be drawn separately. This doesn't change their meaning.



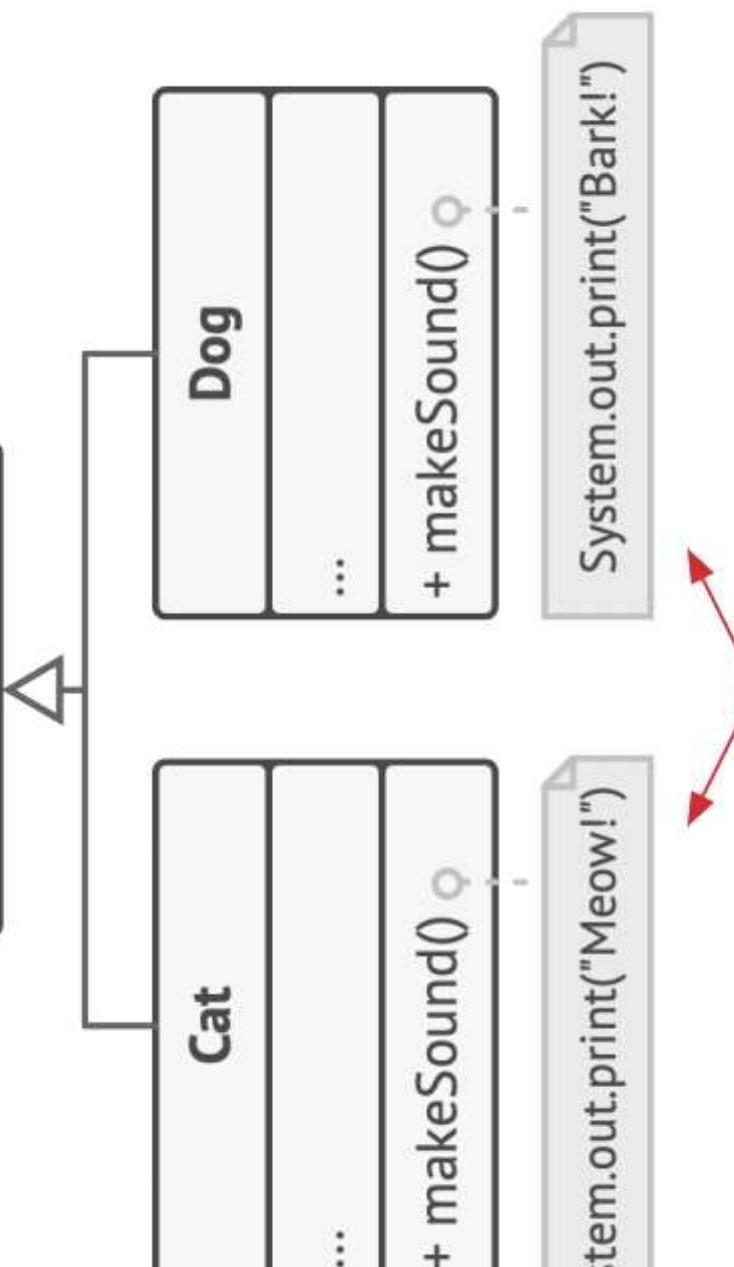
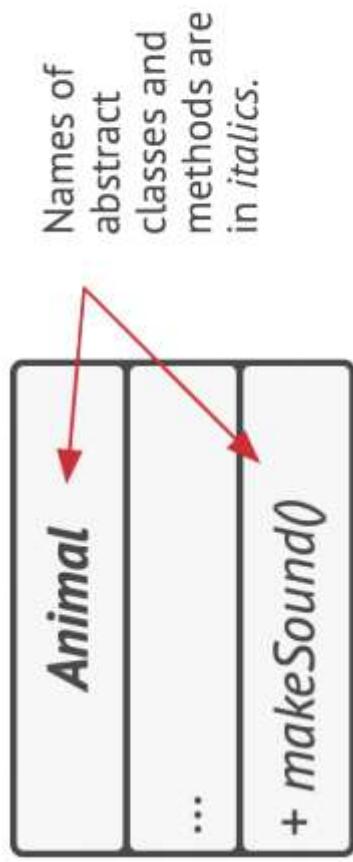
UML diagram of a class hierarchy. All classes in this diagram are part of the **Animal** class hierarchy.

Interfaces



UML diagram of several classes implementing an interface.

Polimorfismo



These are UML comments. Usually they explain implementation details of the given classes or methods.

Dependência

Dependency



- **Dependência:** A classe A pode ser afetada por mudanças na classe B.
- **Associação:** O objeto A conhece o objeto B. A classe A depende de B.
- **Agregação:** O objeto A conhece o objeto B e é composto por B. A classe A depende de B.
- **Composição:** O objeto A conhece o objeto B, é composto por B e gerencia o ciclo de vida de B. A classe A depende de B.
- **Implementação:** A classe A define os métodos declarados na interface B. Os objetos A podem ser tratados como B. A classe A depende de B.
- **Herança:** A classe A herda a interface e a implementação da classe B, mas pode estendê-la. Os objetos A podem ser tratados como B. A classe A depende de B.

Association



Aggregation



Composition



Implementation



Inheritance



Relations between objects and classes: from weakest to strongest

Princípios de projeto

- Impostos são dependentes de diversas condições
- Como resolver?

```
1 method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4         total += item.price * item.quantity
5
6     if (order.country == "US")
7         total += total * 0.07 // US sales tax
8     else if (order.country == "EU"):
9         total += total * 0.20 // European VAT
10
11    return total
```

BEFORE: tax calculation code is mixed with the rest of the method's code.

Separar do uso melhora



```
1 method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4         total += item.price * item.quantity
5
6     total += total * getTaxRate(order.country)
7
8     return total
9
10    method getTaxRate(country) is
11        if (country == "US")
12            return 0.07 // US sales tax
13        else if (country == "EU")
14            return 0.20 // European VAT
15        else
16            return 0
```

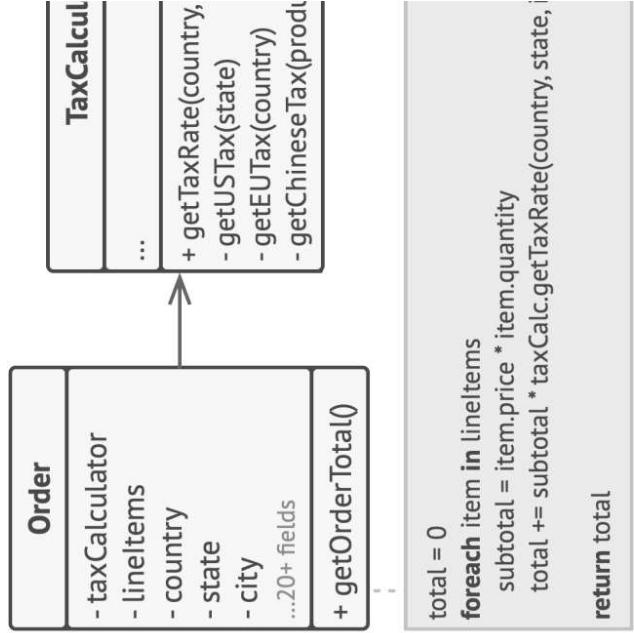
AFTER: you can get the tax rate by calling a designated method.

Mas podemos fazer melhor

Programe para uma interface e não para um implementação!

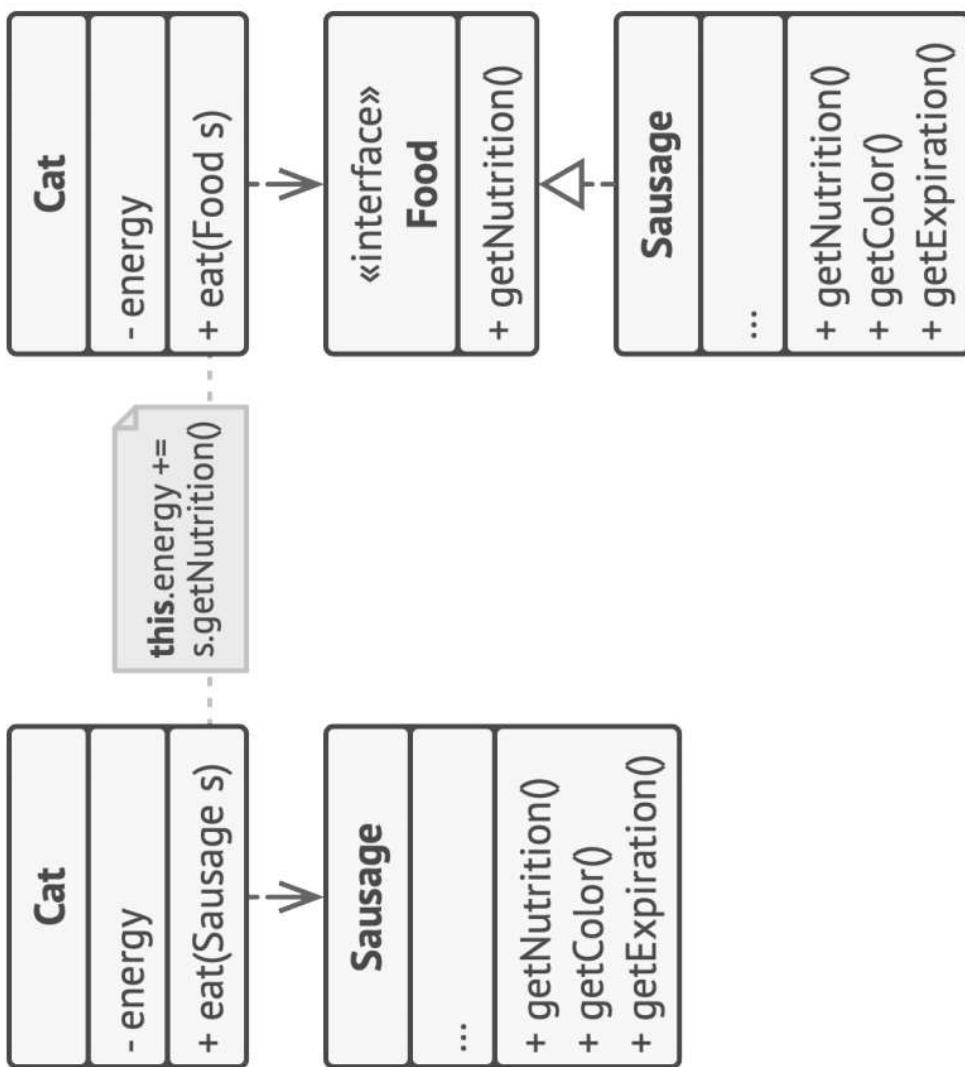
Objects of the `Order` class delegate all tax-related calculations to a `TaxCalculator` object.

BEFORE: calculating tax in `Order` class



Outro exemplo

- Quando fazer isso?



Before and after extracting the interface. The code on the right is more flexible than the code on the left, but it's also more complicated.

Problema de herança



- A herança é provavelmente a maneira mais óbvia e fácil de **reutilizar código entre classes.**
- Você tem **duas classes com o mesmo código. Crie uma classe base comum para essas duas e move o código semelhante para ela. Moleza!**
- Infelizmente, a **herança tem suas ressalvas**, que muitas vezes só se tornam aparentes depois que seu programa já tem muitas classes e mudar qualquer coisa é bem difícil.

Problemas

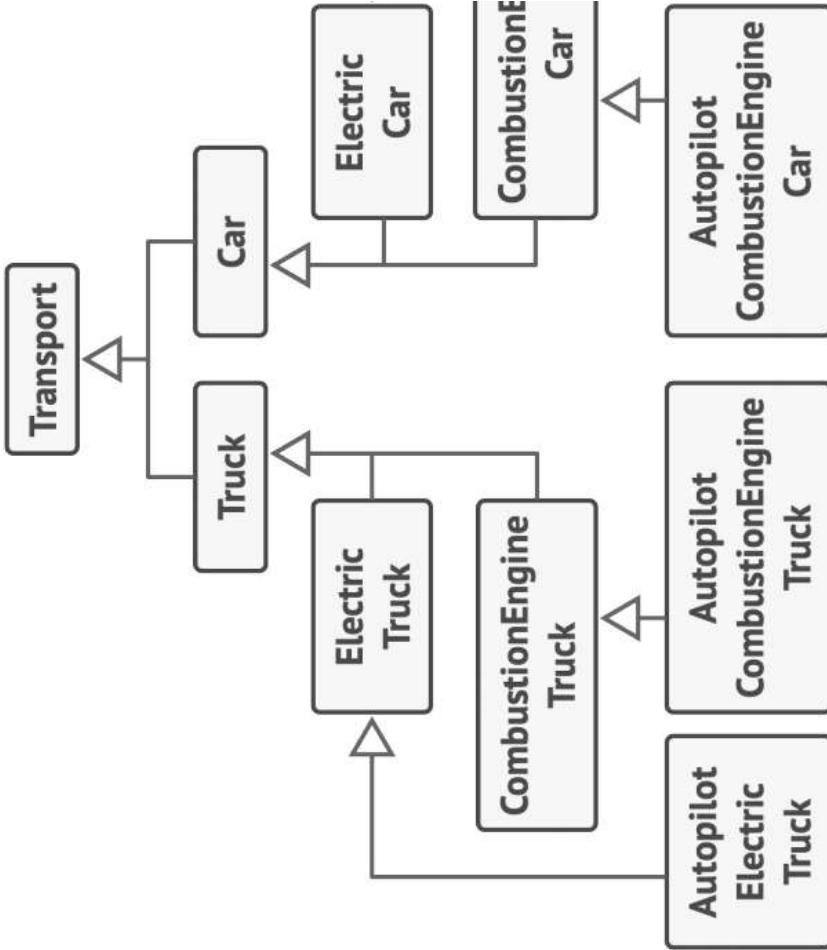
- Uma subclasse **não pode reduzir** a interface da superclasse. Você precisa implementar todos os métodos abstratos da classe pai, mesmo que não vá usá-los.
- Ao sobrescrever métodos, você precisa garantir que o novo **comportamento seja compatível** com o da classe base. Isso é importante porque objetos da subclasse podem ser passados para qualquer código que espera objetos da superclasse e não quer que esse código quebre.

Problemas

- A herança quebra o encapsulamento da superclasse porque detalhes internos da classe pai ficam disponíveis para a subclasse. Pode haver uma situação oposta, em que um programador torna uma superclasse cliente de alguns detalhes das subclasses para facilitar futuras extensões.
- As subclasses são fortemente acopladas às superclasses. Qualquer alteração em uma superclasse pode quebrar a funcionalidade das subclasses.

Problemas

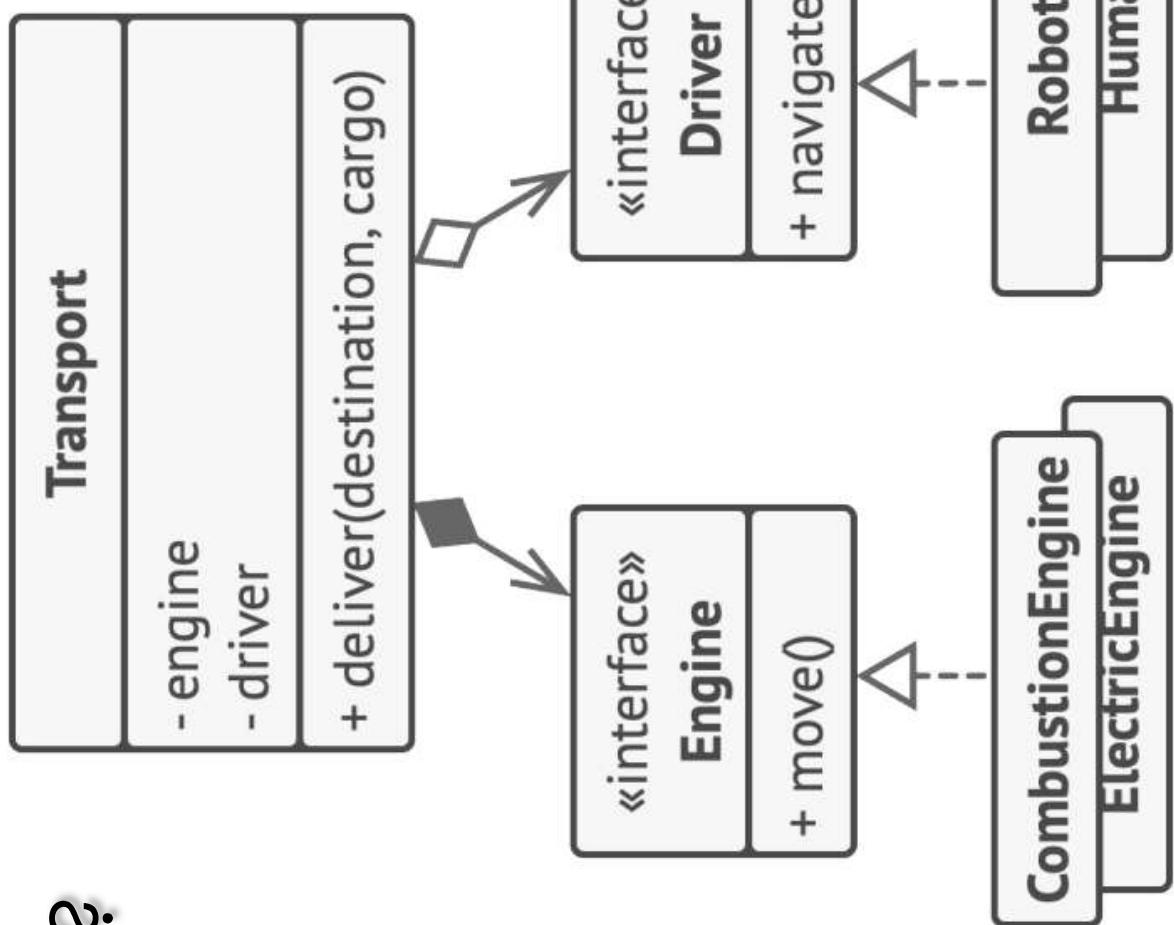
- Tentar reutilizar código por meio de herança pode levar à criação de hierarquias de herança paralelas. A herança geralmente ocorre em uma única dimensão. Mas sempre que há duas ou mais dimensões, você precisa criar muitas combinações de classes, **inchando a hierarquia de classes a um tamanho absurdo.**



INHERITANCE: extending a class in several dimensions (engine type x navigation type) may lead to a combinatorial explosion of subclasses.

Qual alternativa?

- Existe uma alternativa à herança chamada **composição**. Enquanto a herança representa a relação "é um" entre classes (um carro é um meio de transporte), a composição representa a relação "tem um" (um carro tem um motor)



COMPOSITION: different “dimensions” of functionality extend own class hierarchies.

Princípios SOLID



- Agora que você conhece os princípios básicos de design, vamos dar uma olhada em cinco deles, comumente conhecidos como os princípios **SOLID**.
- Robert Martin os apresentou no livro Desenvolvimento Ágil Software: Princípios, Padrões e Práticas¹.
- SOLID é um mnemônico para cinco princípios de design que visam tornar os projetos de software mais compreensíveis, flexíveis e fáceis de manter.

¹ Agile Software Development, Principles, Patterns, Patterns
<https://refactoring.guru/principles-book>

Pense antes de usar!

- Usar esses princípios sem pensar **pode causar mais mal do bem.**
- O custo de aplicar esses princípios à arquitetura de um programa pode ser torná-lo mais complicado do que deve!
- Um programa com poucas linhas e fácil de entender não precisa de engenharia a mais para ser fácil de entender. Isso terá o efeito inverso: mais linhas e mais complicado

Continua na próxima aula



- Vamos ver 2 padrões de projeto e depois continuamos sobre os princípios de programação

The Catalog of Design Patterns

Creational patterns

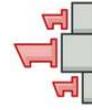
These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.



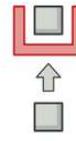
Factory Method
Abstract Factory



Builder
Prototype



Singleton



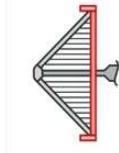
Proxy

Structural patterns

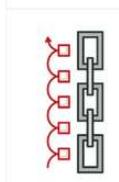
These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



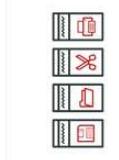
Adapter



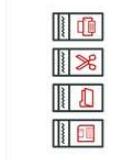
Bridge



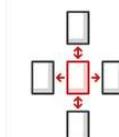
Composite
Chain of Responsibility



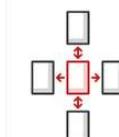
Factory Method
Command



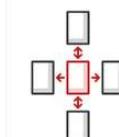
Iterator



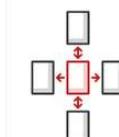
Mediator



Observer
State



Strategy

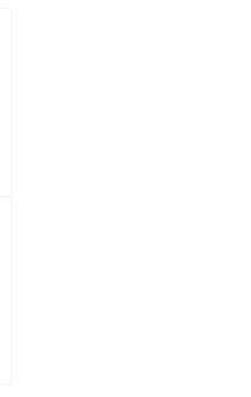
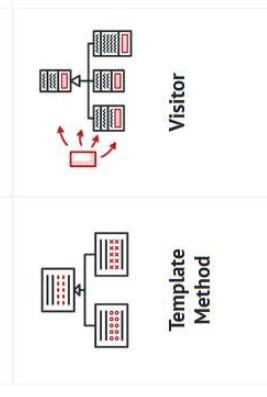
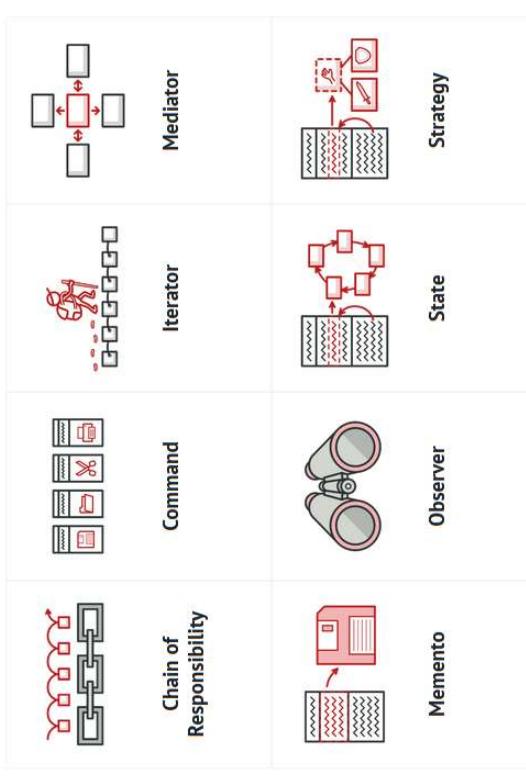


Visitor

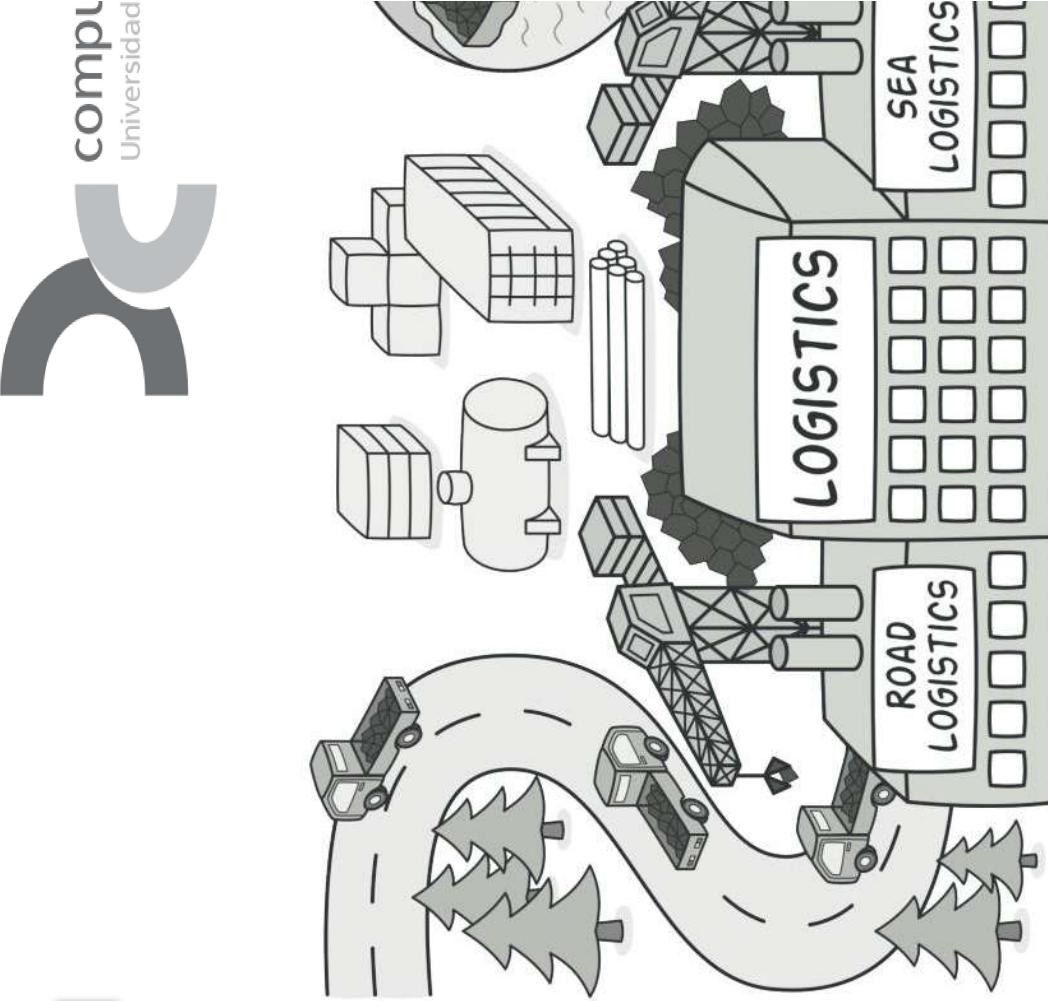
These patterns are concerned with algorithms and the assignment of responsibilities between objects.

Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.



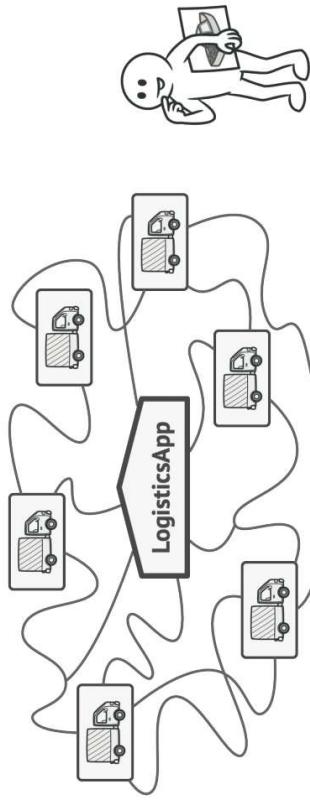
Factory Method



- O **Factory Method** é um padrão de projeto criacional que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.

Problema

- Imagine que você está criando um aplicativo de gerenciamento de logística. A primeira versão do seu aplicativo só pode lidar com transporte por caminhões, então a maior parte do seu código reside na classe Truck.



Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

Problema

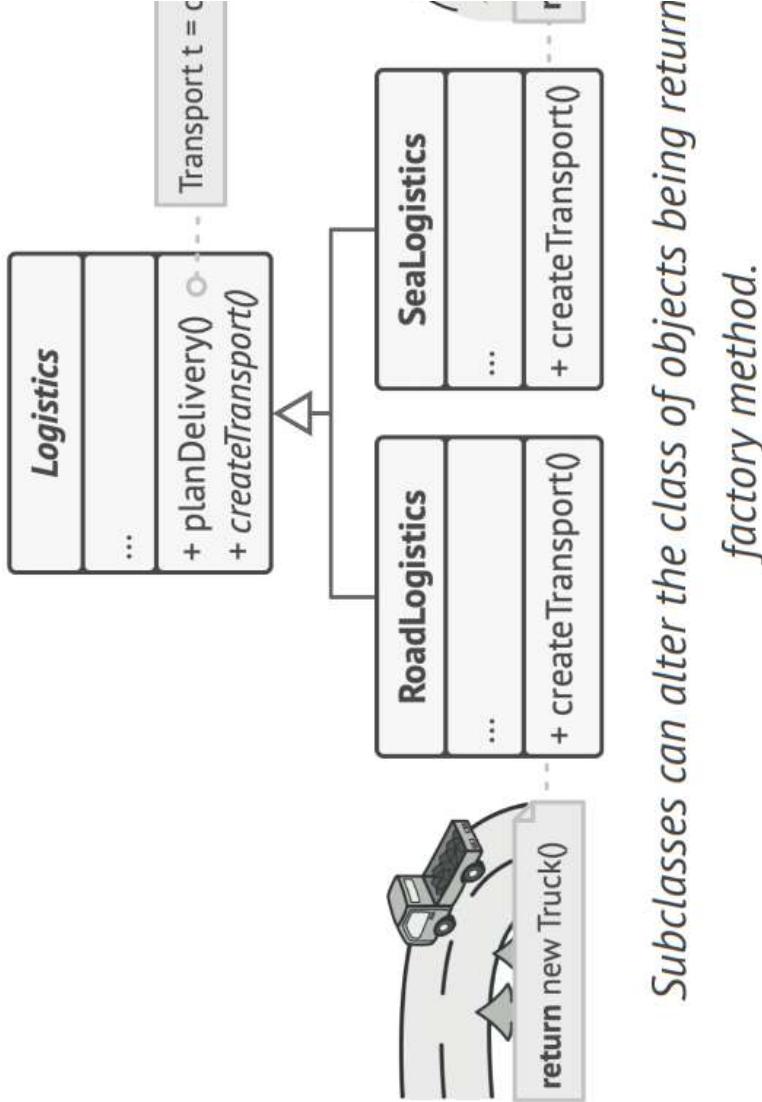
- Depois de um tempo, seu aplicativo se torna bastante popular. Todos os dias você recebe dezenas de solicitações de emprego para incorporar a logística marítima ao aplicativo.
- Ótima notícia, certo? Mas e o código? Atualmente, a maioria parte do seu código está acoplada à classe Truck. **Adicionar Ships ao aplicativo exigiria alterações em toda a base de código.** Além disso, se mais tarde você decidir adicionar outro tipo de transporte ao aplicativo, provavelmente precisará refazer todas essas alterações.

Solução

- O padrão Factory Method sugere que você **substitua as chamadas diretas de construção de objetos** (usando o operador new) por chamadas a um método de fábrica especial.
- Não se preocupe: os objetos ainda são criados por meio do operador new, mas ele está sendo chamado de dentro do método de fábrica. Os objetos retornados por um método fábrica são **frequentemente** chamados de **produtos**.

Solução

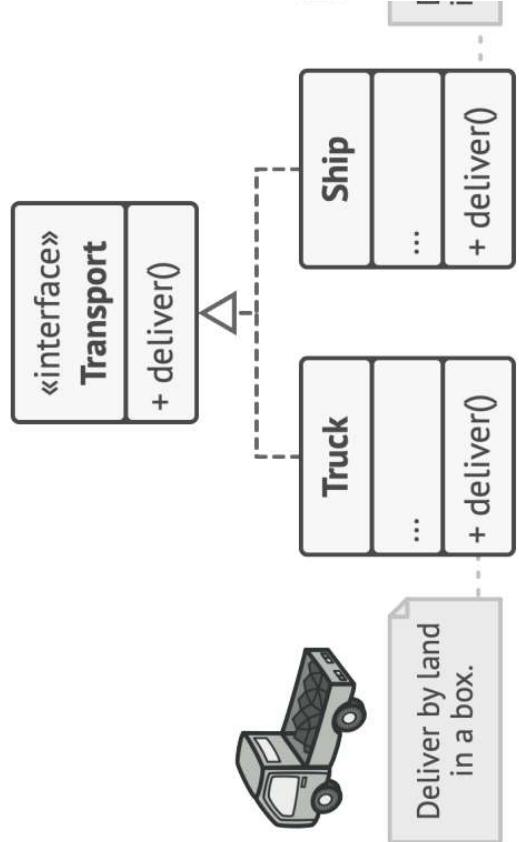
- À primeira vista, essa mudança pode parecer sem sentido: apenas movemos a chamada do construtor de uma parte do programa para outra. No entanto, considere o seguinte: **agora você pode sobrecrever o método de fábrica em uma subclasse e alterar a classe de produtos que estão sendo criados pelo método.**



Subclasses can alter the class of objects being returned by the factory method.

Solução

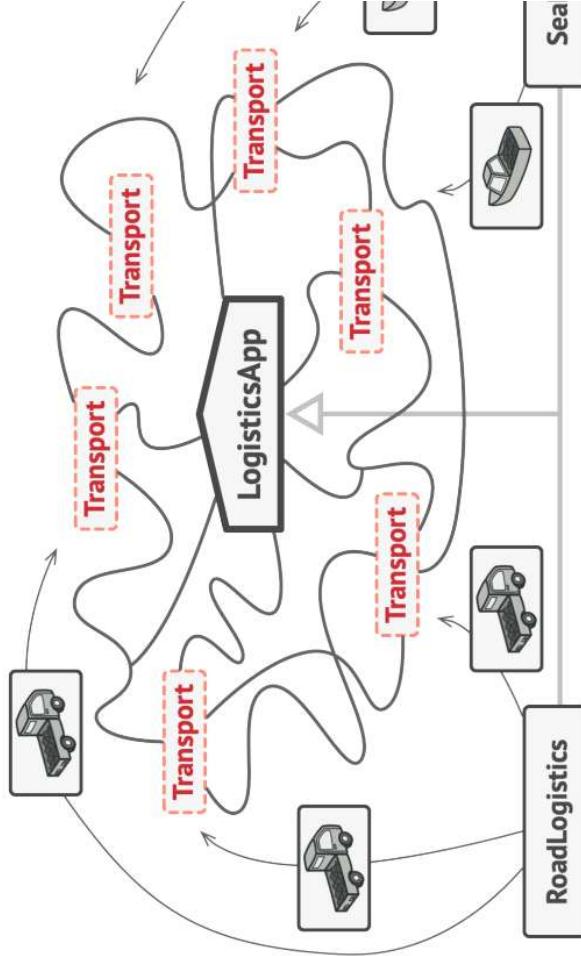
- Por exemplo, as classes Truck e Ship devem implementar a interface Transport, que declara um método chamado deliver. Cada classe implementa esse método de forma diferente: caminhões entregam carga por terra, navios entregam carga por mar. O método de fábrica na classe RoadLogistics retorna objetos de caminhão, enquanto o método de fábrica na classe SeaLogistics retorna navios.



All products must follow the same interface.

Vantagens

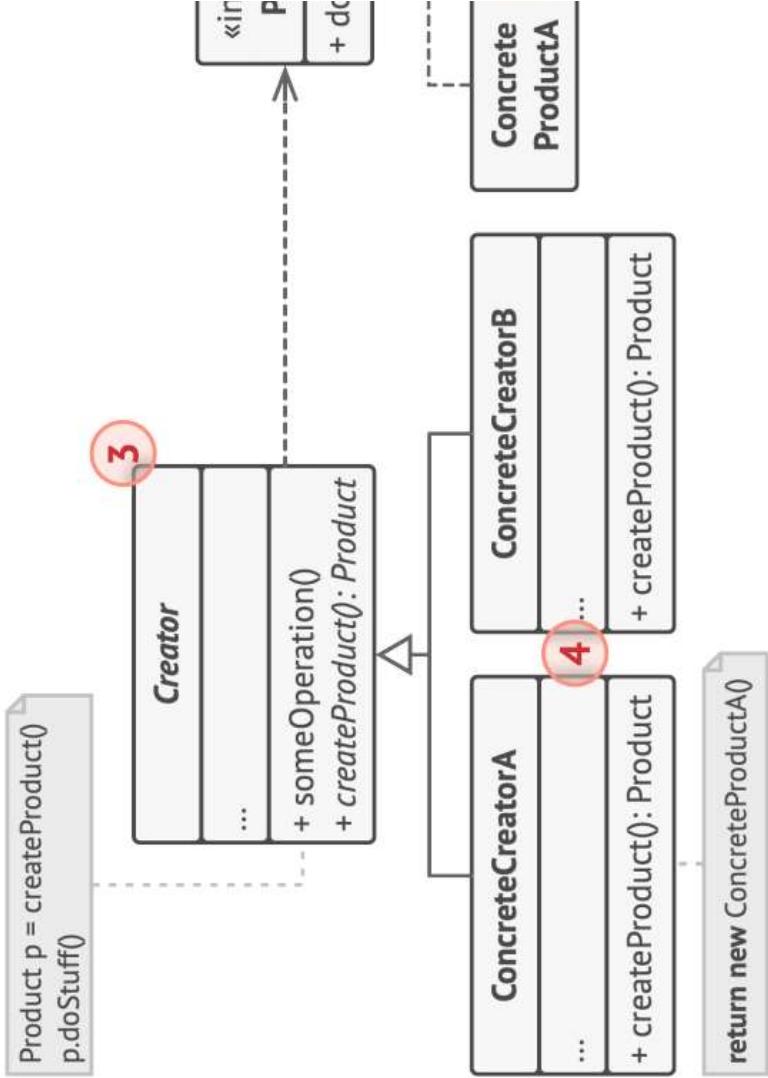
- O código que usa o método de fábrica (frequentemente chamado de código cliente) não vê diferença entre os produtos reais retornados por várias subclasses. O cliente trata todos os produtos como Transporte abstrato.



As long as all product classes implement a common interface, pass their objects to the client code without breaking it

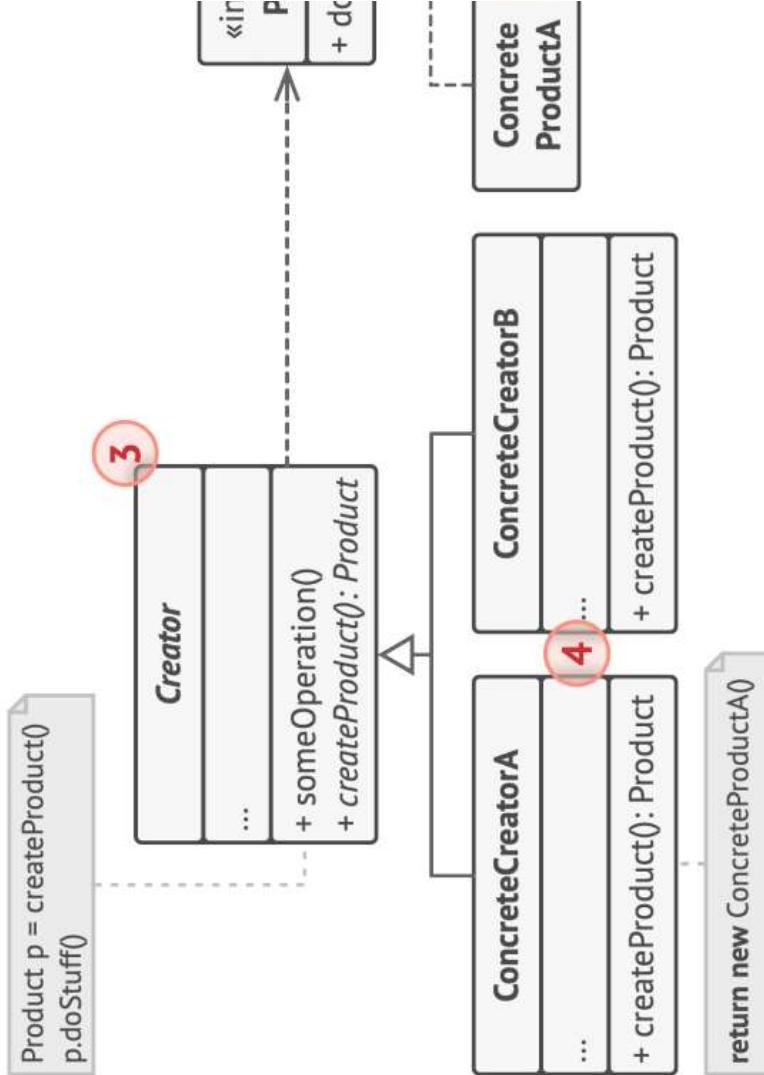
Estrutura

- O **Product** declara a interface, que é comum a todos os objetos que podem ser produzidos pelo criador e suas subclasses.



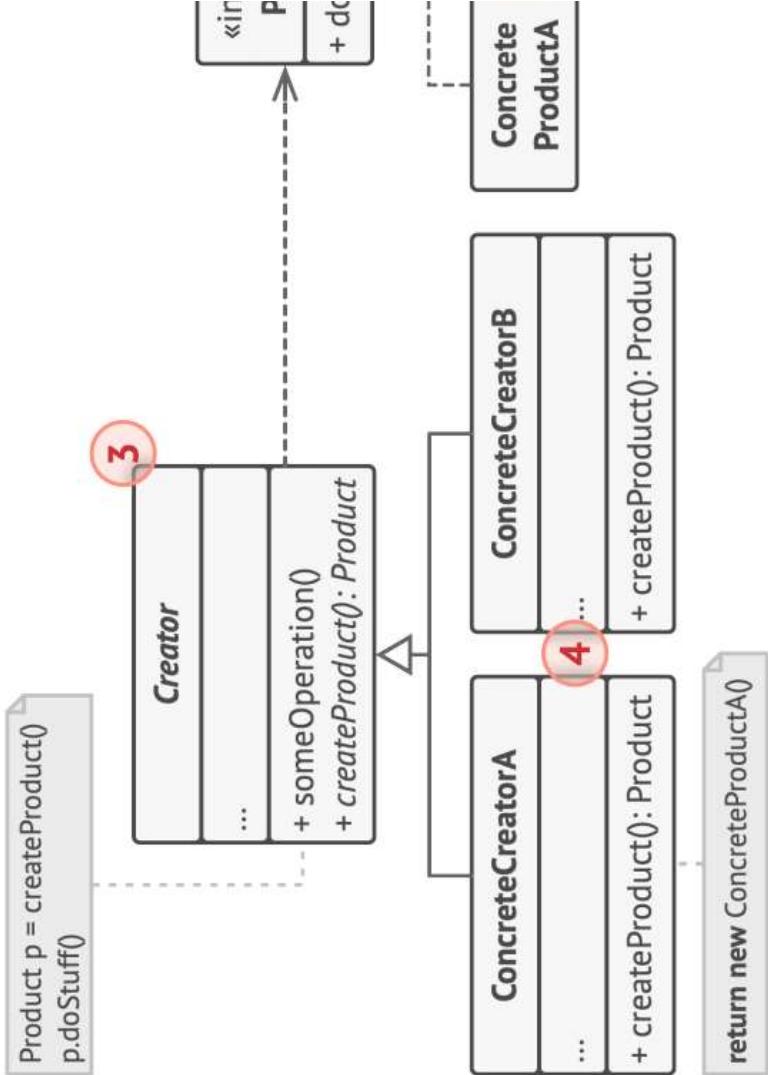
Estrutura

- Os produtos concretos
são diferentes
implementações da
interface do produto.



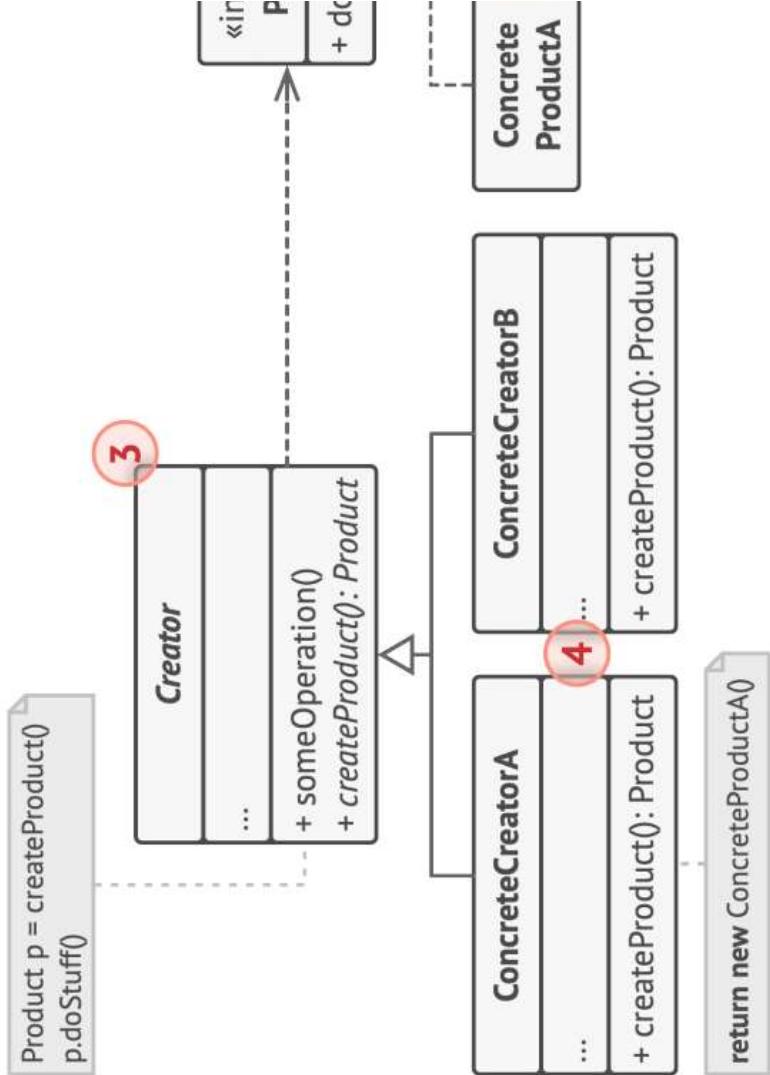
Estrutura

- A classe Creator declara o método de fábrica que retorna novos objetos de produto. É importante que o tipo de retorno deste método corresponda à interface do produto.



Estrutura

- Os **ConcreteCreators** substituem o método de fábrica base para que ele retorne um tipo de produto diferente.



Patrón de diseño

```
1 // The creator class declares the factory method that must  
2 // return an object of a product class. The creator's subclasses  
3 // usually provide the implementation of this method.  
4 class Dialog is  
5   // The creator may also provide some default implementation  
6   // of the factory method.  
7 abstract method createButton():Button  
8  
9   // Note that, despite its name, the creator's primary  
10  // responsibility isn't creating products. It usually  
11  // contains some core business logic that relies on product  
12  // objects returned by the factory method. Subclasses can  
13  // indirectly change that business logic by overriding the  
14  // factory method and returning a different type of product  
15  // from it.  
16 method render() is  
17   // Call the factory method to create a product object.  
18 Button okButton = createButton()  
19   // Now use the product.  
20 okButton.onClick(closedDialog)  
21 okButton.render()
```

Pseudocode

```
24 // Concrete creators override the factory method to change the
25 // resulting product's type.
26 class WindowsDialog extends Dialog is
27     method createButton():Button is
28         return new WindowsButton()
29
30 class WebDialog extends Dialog is
31     method createButton():Button is
32         return new HTMLButton()
33
34 // The product interface declares the operations that all
35 // concrete products must implement.
36 interface Button is
37     method render()
38     method onClick(f)
39
40 // Concrete products provide various implementations of the
41 // product interface.
42
```

Pseudocode

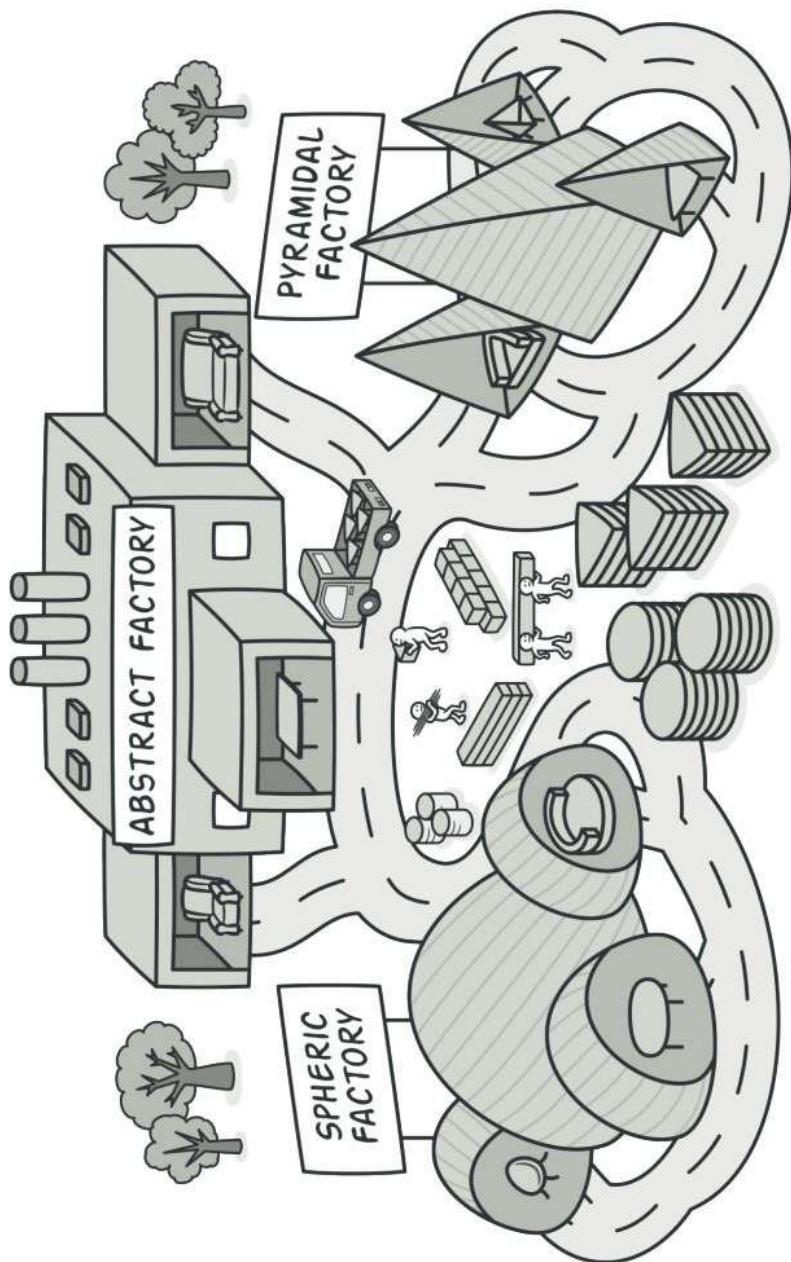
```
56 class Application is
57   field dialog: Dialog
58
59   // The application picks a creator's type depending on the
60   // current configuration or environment settings.
61   method initialize() is
62     config = readApplicationConfigFile()
63
64   if (config.OS == "Windows") then
65     dialog = new WindowsDialog()
66   else if (config.OS == "Web") then
67     dialog = new WebDialog()
68   else
69     throw new Exception("Error! Unknown operating system.")
70
71   // The client code works with an instance of a concrete
72   // creator, albeit through its base interface. As long as
73   // the client keeps working with the creator via the base
74   // interface, you can pass it any creator's subclass.
```



Aplicação



- Use quando você não souber de antemão os tipos e dependências exatos dos objetos com os quais seu código deve trabalhar.
- Use quando quiser fornecer aos usuários da sua biblioteca framework uma maneira de entender seus componentes internos.
- Use quando quiser economizar recursos do sistema reutilizando objetos existentes em vez de recriá-los a cada

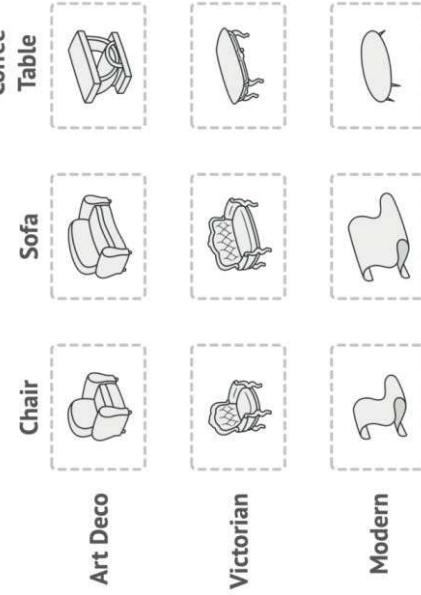


ABSTRACT FACTORY

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Problema

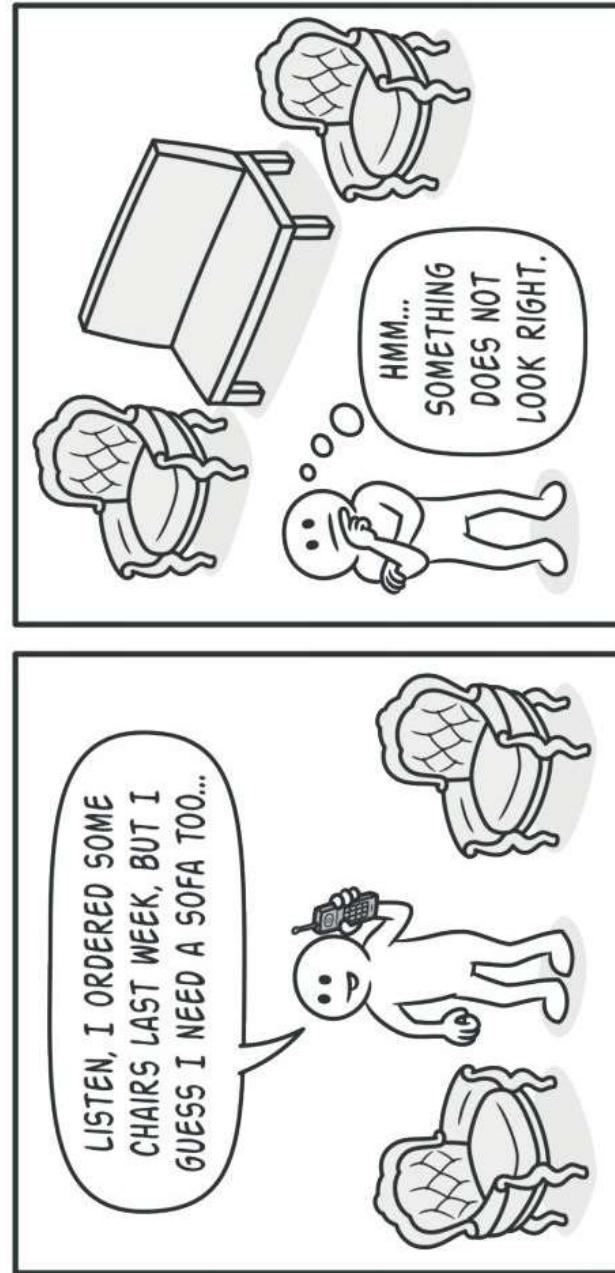
- Imagine que você está criando um simulador de loja de móveis. Seu código consiste em classes que representam:
 - Uma família de produtos relacionados, por exemplo: Cadeira + Sofá + Mesa de centro.
 - Existem diversas variantes desta família. Por exemplo, os produtos Cadeira+ Sofá + Mesa de Centro estão disponíveis nestas variantes Moderna, Vitoriana, Art Déco.



Product families and their variants.

Problema

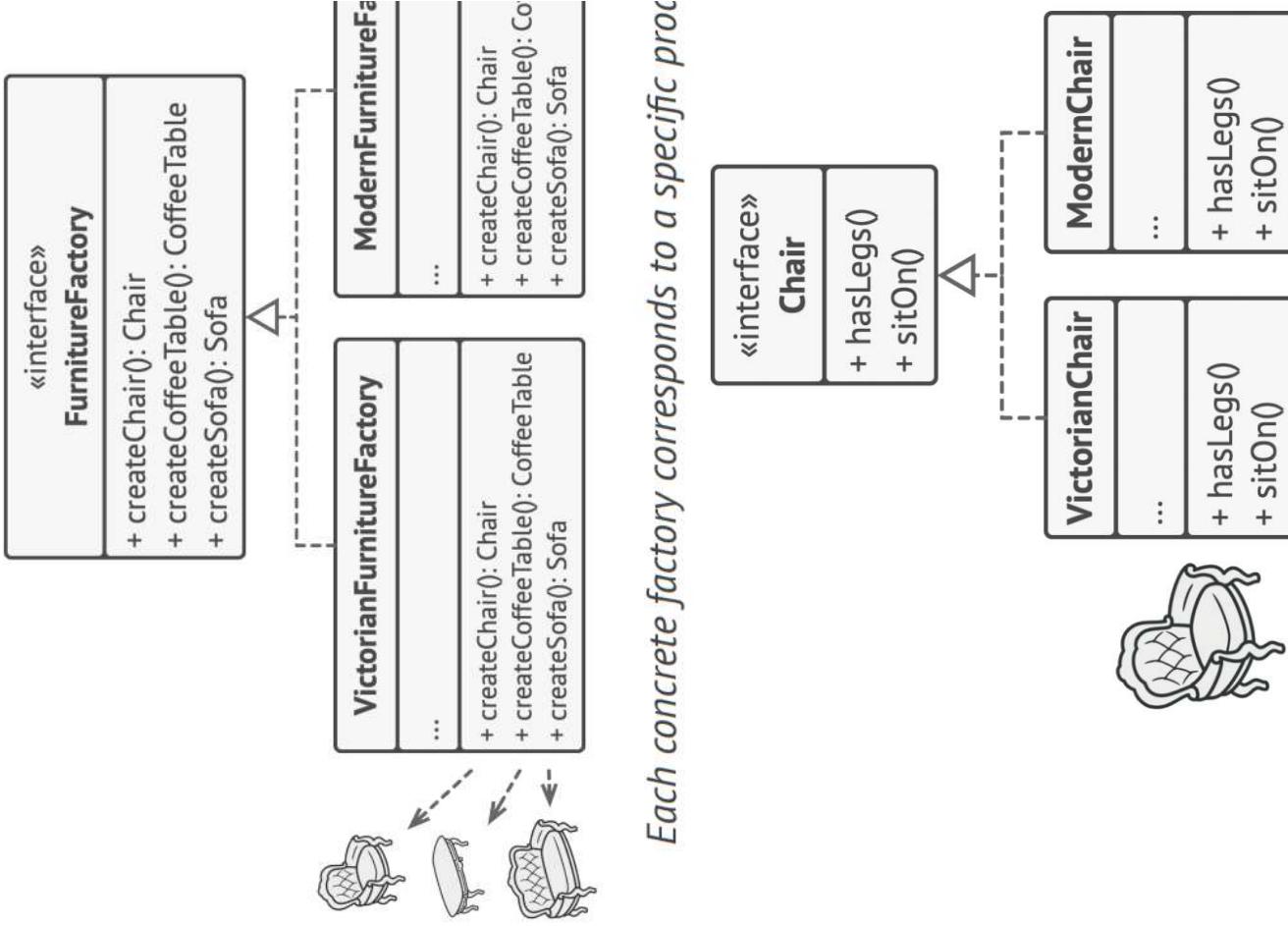
- Garantir que todos os móveis sejam do mesmo tipo após definido o tipo



A Modern-style sofa doesn't match Victorian-style chairs.

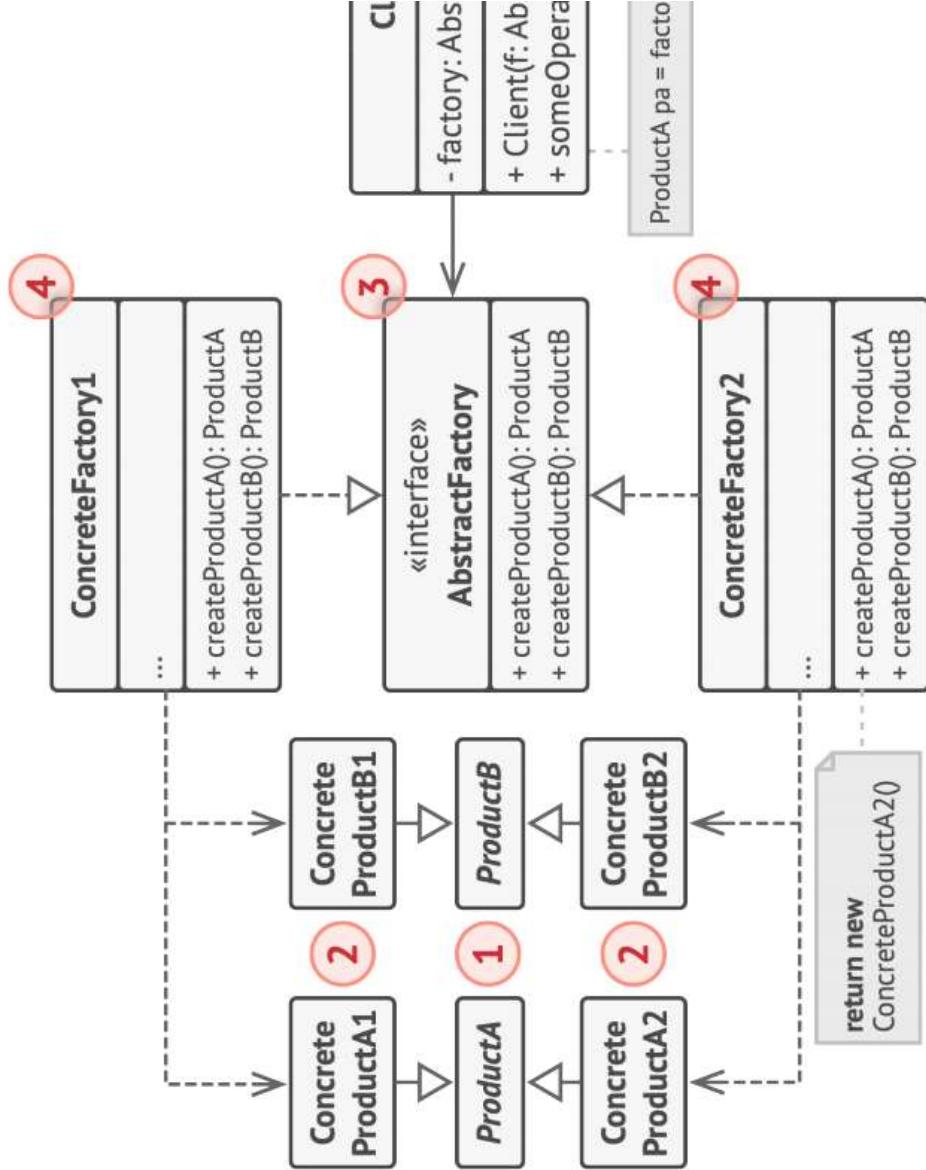
Solução

- O próximo passo é declarar a Fábrica Abstrata — uma interface com uma lista de métodos de criação para todos os produtos que fazem parte da família de produtos.
- Esses métodos devem retornar tipos abstratos de produtos.



Estrutura

- A escolha da classe é feita uma vez.
- Não existe uma cadeia de IFs
- A AbstractFactory é basicamente um gerenciador de ConcreteFactory



Aplicações

- Use a Abstract Factory quando seu código precisar funcionar **com várias famílias de produtos relacionados**, mas você não quiser que ele dependa das classes concretas desses produtos
 - elas podem ser desconhecidas antecipadamente ou você simplesmente deseja permitir extensibilidade futura.
- Por exemplo, se no nosso trabalho anterior de navegação fosse usado para um jogo então cada ambiente diferente teríamos modelos compatíveis em todas as fábricas quando nível está sendo jogado

Trabalho 6: Fábricas

- **Alterar o trabalho 5 para usar fábricas**

- Torne o projeto orientado a objetos
- Faça um tipo de fábrica para cada grupo de coisas similares, por exemplo
 - Origem e destino (podemos ter um destino dinâmico depois)
 - Caminho só horizontal e vertical e outro que permite diagonal
 - Obstáculos (quais tipos?)
 - Grid (quais tipos?)
 - Algoritmos (quais tipos?)
 - Etc
- Note que temos um conjunto de coisas que estão ligadas ao grid e temos outro conjunto de coisas que estão ligadas ao algoritmos (etc), nisso entra a fábrica abstrata
- Queremos chegar no que está na figura, então prepare o código para isso.
- Outro nível do uso da fábrica é tornar toda a parte gráfica multiplataforma, não precisa fazer isso, mas faria isso e me diga na próxima aula quando for na sua mesa ver o trabalho

- **GIT**

- Código
- **Prazo: 18/11 23:59**

